Geometric Modeling

Assignment 3: Discrete Differential Quantities

Julian Panetta -- fjp234@nyu.edu

Acknowledgements: Olga Diamanti

NYU | COURANT

Assignment 3: Discrete Normals, Curvatures, and Smoothing

Handout date: 03/08/2017
Submission deadline: None
Demo session: None

Note: This assignment is optional and won't be graded. But next assignment assumes familiarity with differential operators (in particular, the discrete Laplacian), so we encourage you to work through it.

In this exercise you will

- Experiment with different ways to compute surface normals for triangle meshes.
- Calculate curvatures from a triangle mesh.
- Perform mesh smoothing.
- Familiarize yourselves with the relevant implementations in `libigl`.

1. Vertex Normals

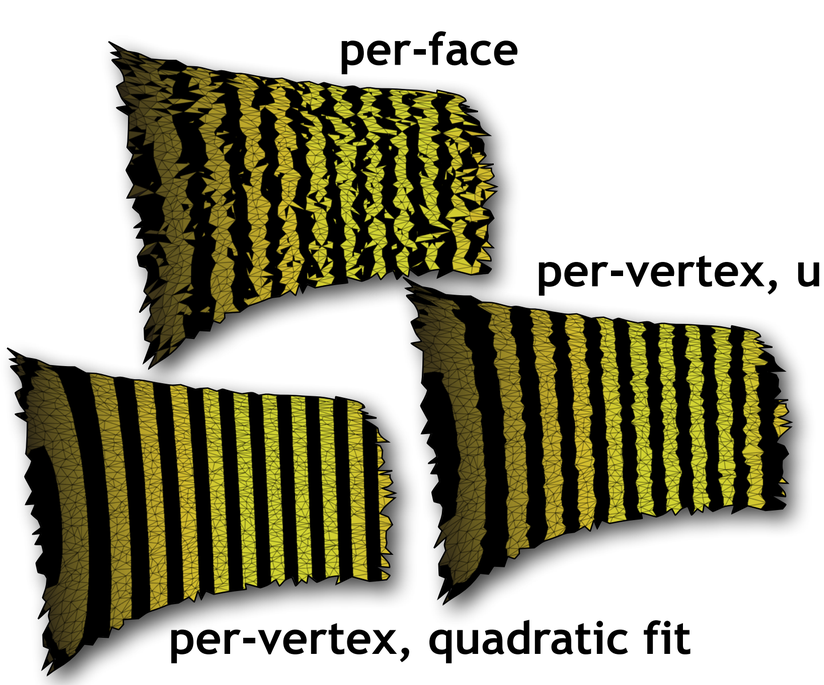
Starting from the `libigl` tutorial, experiment with different ways to compute per-vertex normals. A good starting point is tutorial # 201 of and the documentation inside function `igl::per_vertex_normals`. Also check tutorial # 205 for the calculation of the discrete Laplacian. You will need to compute and switch between these different normals:

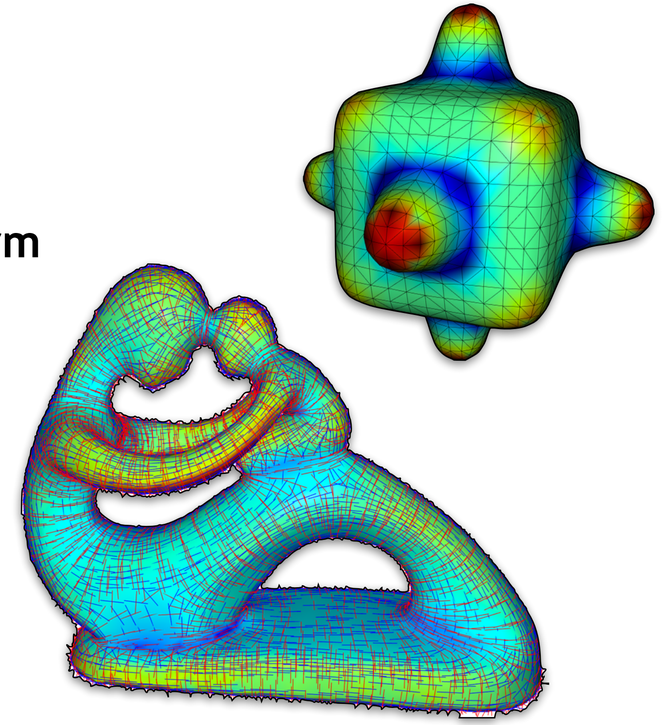
- **Standard vertex normals** These are computed as the unweighted average of surrounding face normals.
- **Area-Weighted normals** Same as above, but the average is weighted by face area.
- **Mean-curvature normals** Apply the cotangent-weighted Laplacian to the mesh vertex positions to compute the normal weighted by mean curvature as shown in class. If this vector field's magnitude at a particular vertex is greater than some threshold, ϵ, use the normalized vector as the vertex's normal. Otherwise, the surface is essentially flat at the vertex and the normal can be computed using the area weighted average above. These are the cotangent-weighted discrete Laplacian at every vertex.
- **PCA computation** At each vertex v_i, fit a plane to the k nearest neighbors (with k configurable by GUI) using Principal Component Analysis. The vertex normal is then the principal component with the smallest eigenvalue (the normal to the plane). The neighbours can be collected by running breadth-first search.
- **Normals based on quadratic fitting** Using the local frame computed at a given vertex with PCA as above, the positions of the vertex and its k neighbours can be represented as $[u, v, f(u, v)]$ using a height field function $f(u, v)$. This vector is expressed in the local frame's basis (i.e. the principal component vectors), with u and v giving the tangential coordinates and $f(u, v)$ giving the height. Recall, the height axis is the principal component with smallest eigenvalue. Compute the vertex normal by (a) fitting a quadratic bi-variate polynomial to the

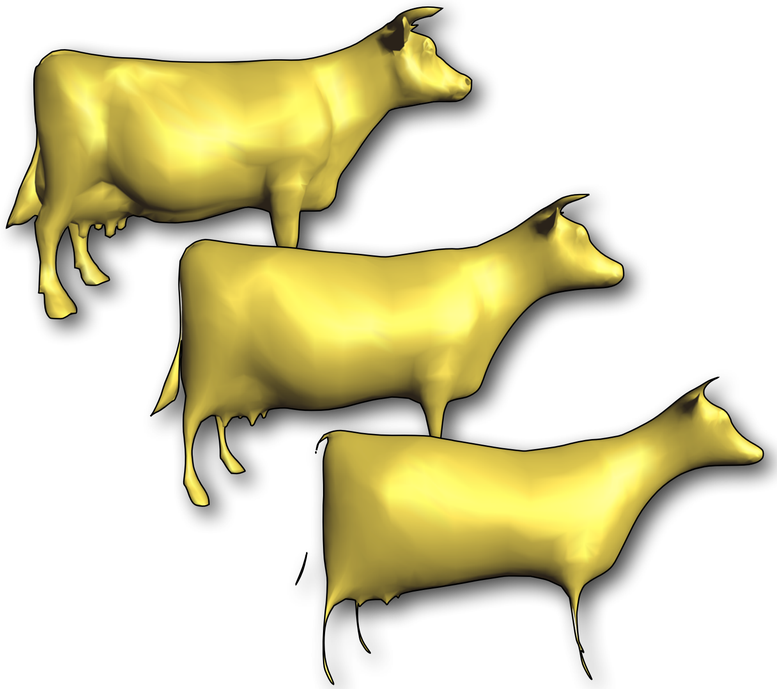
Daniele Panozzo, Julian Panetta
March 8, 2017

Assignment 3 (Optional)

- Topic: Discrete Differential Quantities with libigl
  - Vertex Normals, Curvature, Smoothing (Tutorials 201-205)







Vertex Normals

uniform average

area-weighted average

mean-curvature based

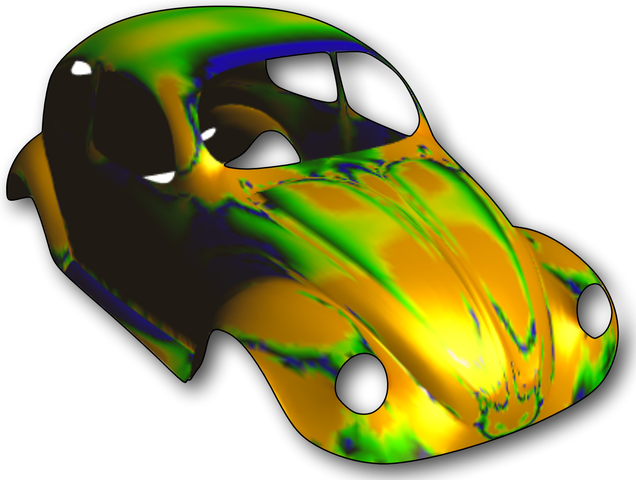
PCA on k-nearest neighbors

quadratic fitting on k-nearest neighbors

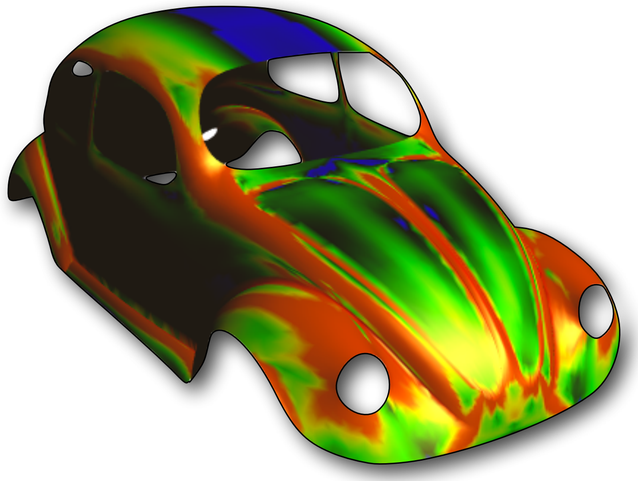
libigl tutorial #201,
igl::principal_curvature

Curvature

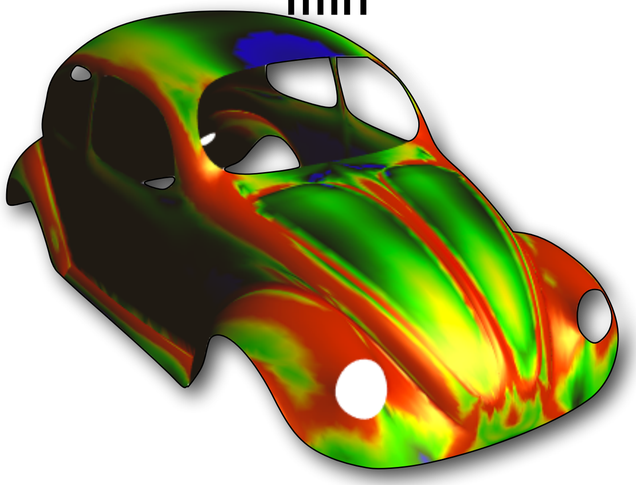
libigl tutorials
#202, #203



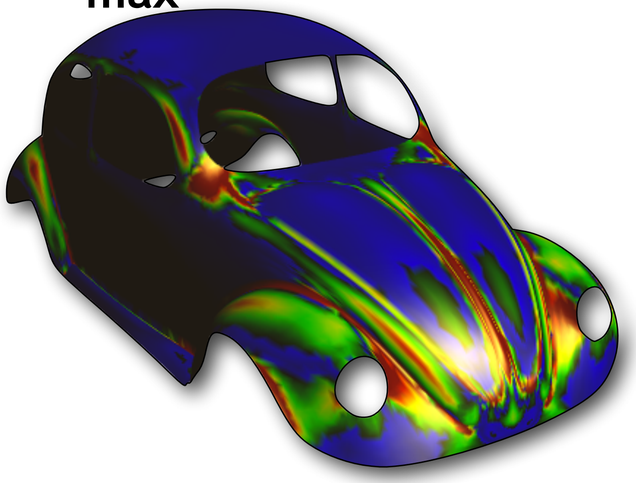
min



max



mean



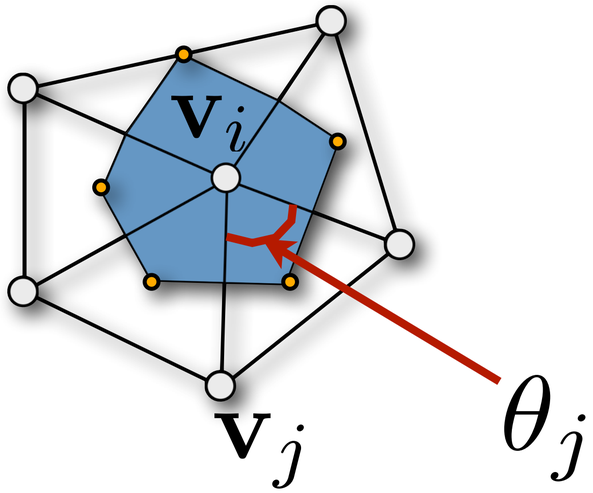
gauss

Curvature

$$\Delta_S \mathbf{p} = -2H\mathbf{n}$$

- Mean Curvature $H(\mathbf{v}_i) = 0.5\|L_c(\mathbf{v}_i)\|$

- Gaussian Curvature $G(\mathbf{v}_i) = \dfrac{2\pi - \sum_j \theta_j}{A(\mathbf{v}_i)}$



- Min Curvature $\kappa_1 = H + \sqrt{H^2 - G}$
- Max Curvature $\kappa_1 = H - \sqrt{H^2 - G}$

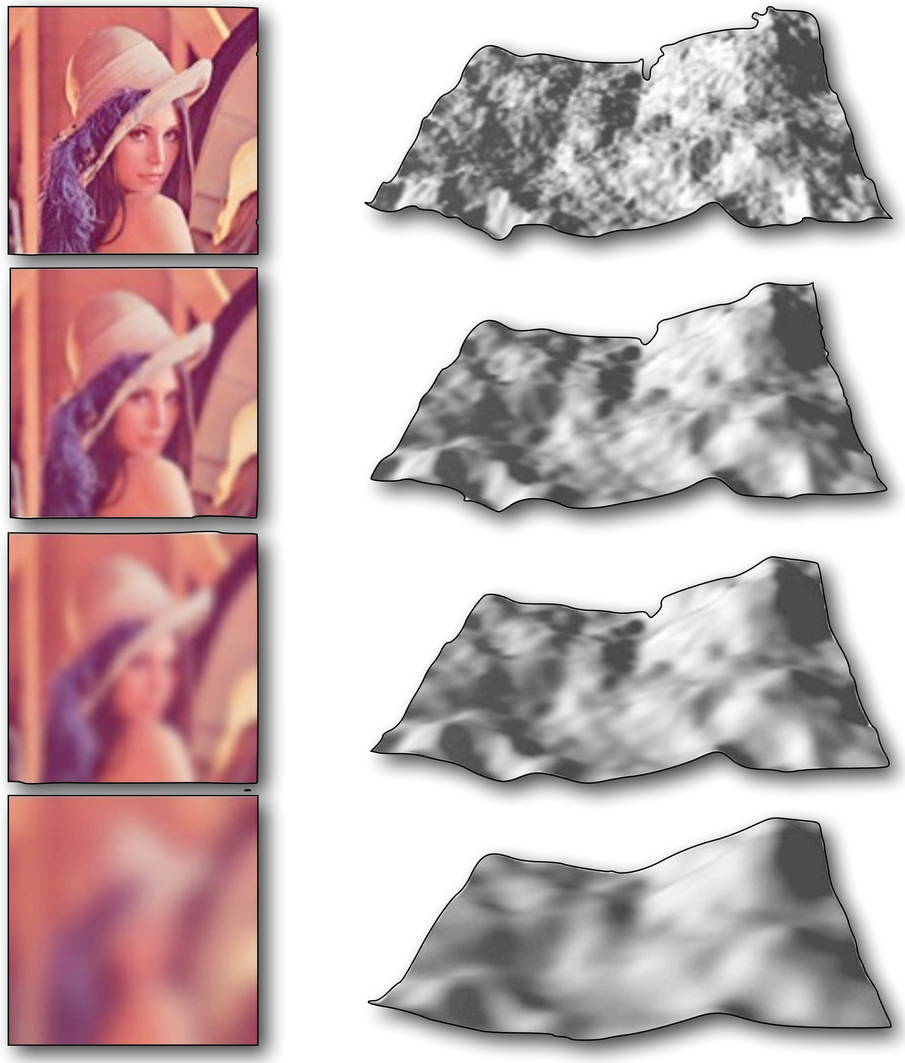
$$G = \kappa_1 \kappa_2$$
$$2H = \kappa_1 + \kappa_2$$

Data Smoothing: Grid Case

- Image/2D data smoothing:
  Filter out noise/rapid oscillations
- Solve the heat equation over some time period:

$$\frac{\partial f}{\partial t} = \lambda \Delta f = \lambda \left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \right)$$

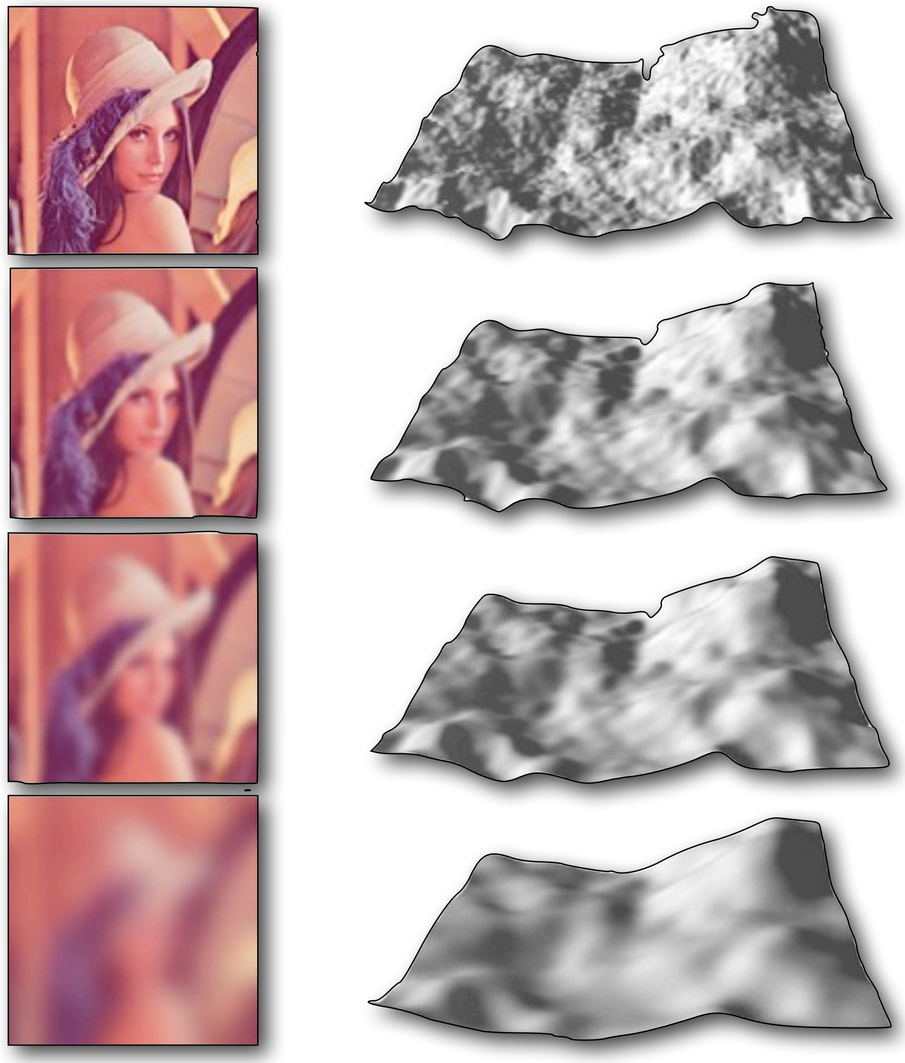
- Intuition: move toward the average of neighboring values



Data Smoothing: Grid Case

$$\frac{\partial f}{\partial t} = \lambda \Delta f = \lambda \left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \right)$$

- Approach: discretize laplacian with **finite differences**
- Now we have an ordinary differential equation (ODE)
- Integrate (time-step) the ODE, e.g., with forward/backwards Euler



Data Smoothing: Surface Case

- Laplace-Beltrami operator lets us smooth functions on surfaces, S

$$\frac{\partial f}{\partial t} = \lambda \Delta_S f$$

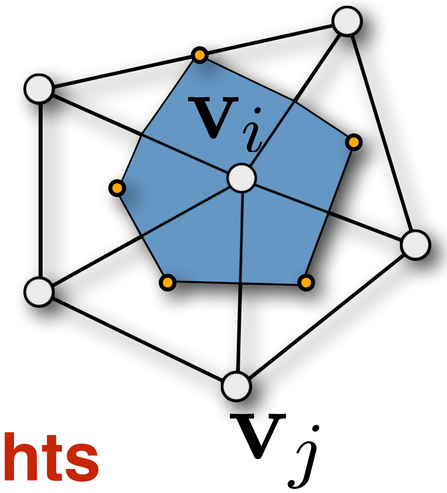
- E.g., with forward Euler (explicit smoothing) on a triangle mesh, M:

$$f^{t+\mathrm{d}t}(\mathbf{v}_i) = f^t(\mathbf{v}_i) + \lambda \, \mathrm{d}t \, \Delta_M f^t(\mathbf{v}_i)$$

$$= f^t(\mathbf{v}_i) + \lambda \, \mathrm{d}t \left(\sum_{\mathbf{v}_j \in N(\mathbf{v}_i) \cup \{\mathbf{v}_i\}} w_{ij} f^t(\mathbf{v}_j) \right)$$

vertices adjacent v_i

discrete Laplacian weights



Mesh Smoothing

- We can view the vertex positions as a (vector-valued) function to smooth!

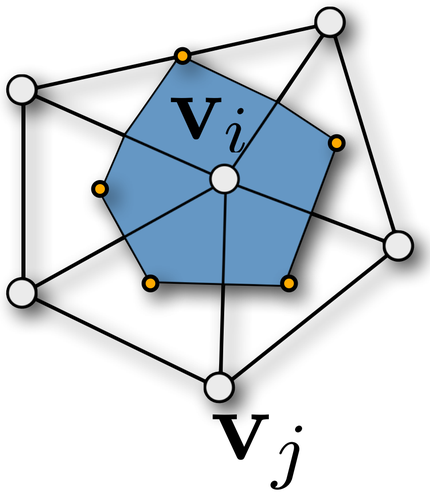
$$f(\mathbf{v}_i) = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix}$$

- Here, first-order explicit time stepping looks like:

$$f^{t+\mathrm{d}t}(\mathbf{v}_i) = f^t(\mathbf{v}_i) + \lambda \, \mathrm{d}t \, \Delta_M f^t(\mathbf{v}_i) \quad \Longrightarrow$$

$$\mathbf{v}_i^{\mathrm{new}} = \mathbf{v}_i + \lambda \, \mathrm{d}t \, (\Delta_M \mathbf{v})_i$$

$$= \mathbf{v}_i + \lambda \, \mathrm{d}t \left(\sum_{\mathbf{v}_j \in N(\mathbf{v}_i) \cup \{\mathbf{v}_i\}} w_{ij} \mathbf{v}_j \right)$$

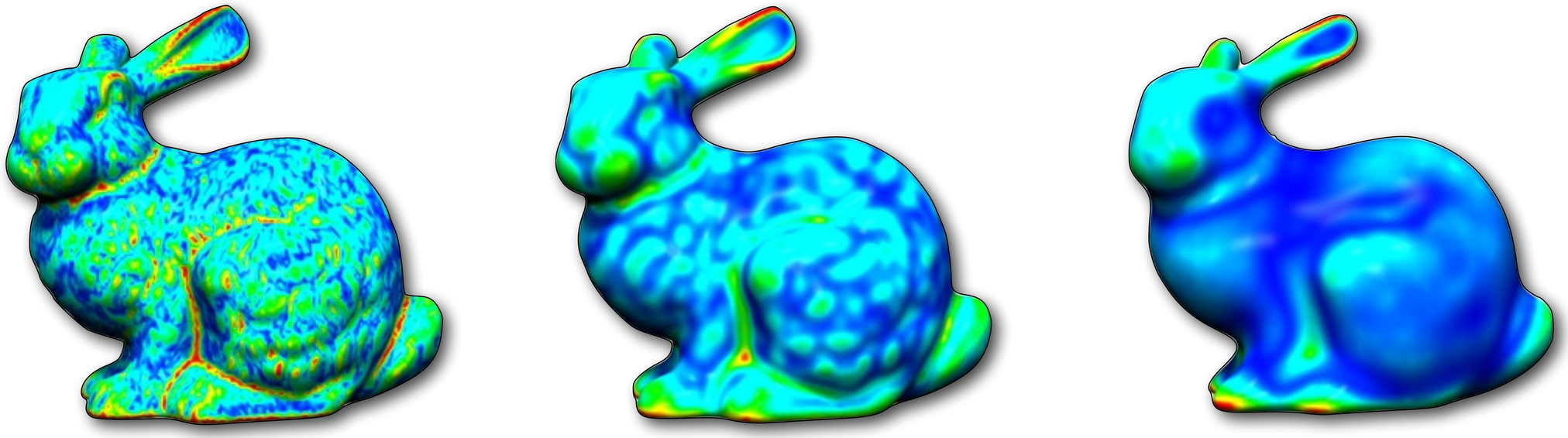


- Note: Laplacian weights generally change when mesh changes!

Explicit Smoothing

- In matrix form: $\frac{\partial \mathbf{v}}{\partial t} = \lambda \mathbf{L}(\mathbf{v})$
- Intuition: vertices moving toward neighbor average.

$$\mathbf{v}^{n+1} - \mathbf{v}^n = \lambda dt \mathbf{L}\mathbf{v}^n \qquad \mathbf{v}^{n+1} = (I + \lambda dt \mathbf{L})\mathbf{v}^n$$



Implicit Smoothing

- "Backward Euler" is more stable
- Take larger time steps without mesh "blowing up" (becoming jagged)

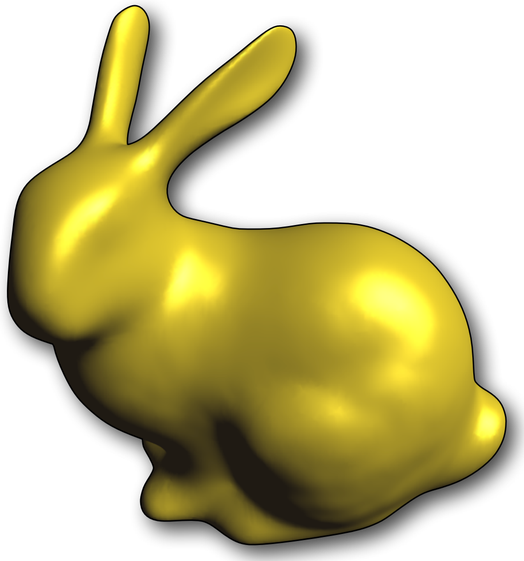
Laplacian at next (unknown) step!

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \lambda dt \mathbf{L}\mathbf{v}^{n+1}$$

Must solve a linear system:

$$(I - \lambda dt \mathbf{L})\mathbf{v}^{n+1} = \mathbf{v}^n$$

original

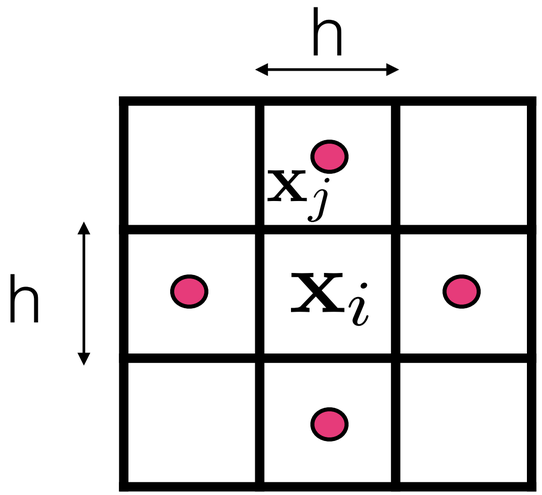


explicit, 1000 iterations, λ = 0.01

implicit, 1 iteration, λ = 20

libigl tutorial #205

2D Rectangular Grid Laplacian



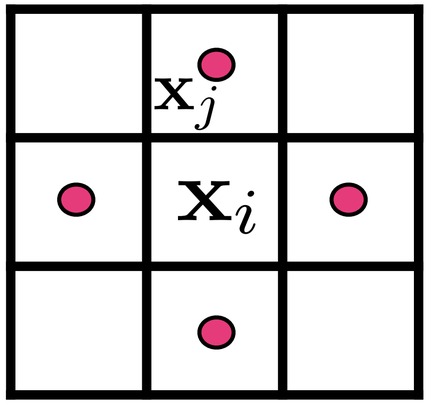
$$\Delta_h f(\mathbf{x}_i) := \frac{1}{h^2}\left[\left(\sum_{\mathbf{x}_j \in N(\mathbf{x}_i)} f(\mathbf{x}_j)\right) - 4f(\mathbf{x}_i)\right]$$

Measures difference from neighbor's average (if we divide by 4/h^2):

Weights can be represented by a "stencil:"

	1	
1	-4	1
	1	

2D Rectangular Grid Laplacian



$$\Delta_h f(\mathbf{x}_i) := \frac{1}{h^2}\left[\left(\sum_{\mathbf{x}_j \in N(\mathbf{x}_i)} f(\mathbf{x}_j)\right) - 4f(\mathbf{x}_i)\right]$$

	1	
1	-4	1
	1	

Derivation:

Compute second derivatives by finite-differencing the forward difference approximation to first derivative:

$$\frac{\partial^2 f}{\partial x^2}(x,y) \approx \frac{1}{h}\left(\frac{f(x+h,y)-f(x,y)}{h} - \frac{f(x,y)-f(x-h,y)}{h}\right)$$
$$= \frac{f(x+h,y)+f(x-h,y)-2f(x,y)}{h^2} \quad \Longrightarrow$$

Sum second partial derivative approximations for each dimension:

$$\Delta f(x,y) = \frac{\partial^2 f}{\partial x^2}(x,y) + \frac{\partial^2 f}{\partial y^2}(x,y)$$
$$\approx \boxed{\frac{f(x+h,y)+f(x-h,y)+f(x,y+h)+f(x,y-h)-4f(x,y)}{h^2}}$$

Irregular Grid (Mesh) Laplacian

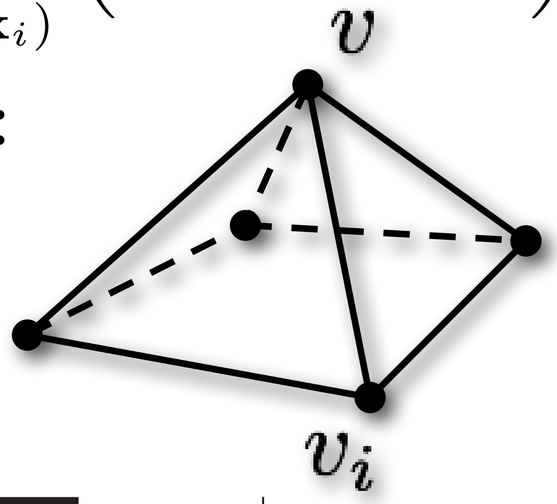
- First approach: “umbrella operator”
  - assume all edges are unit length
  - depends only on connectivity, not vertex locations
- Notice, **when edge length h = 1** the **2D grid** laplacian is:

$$\Delta_h f(\mathbf{x}_i) = \frac{1}{h^2}\left[\left(\sum_{\mathbf{x}_j \in N(\mathbf{x}_i)} f(\mathbf{x}_j)\right) - 4f(\mathbf{x}_i)\right] = \sum_{\mathbf{x}_j \in N(\mathbf{x}_i)} \Big(f(\mathbf{x}_j) - f(\mathbf{x}_i)\Big)$$

- This version easily generalizes to an arbitrary mesh, M:

$$\Delta_M^{\text{uniform}} f(\mathbf{v}) = \sum_{\mathbf{v}_j \in N(\mathbf{v})} \Big(f(\mathbf{v}_j) - f(\mathbf{v})\Big)$$

$$= \left(\sum_{\mathbf{v}_j \in N(\mathbf{v})} f(\mathbf{v}_j)\right) - |N(\mathbf{v})| f(\mathbf{v})$$



#neighbors

The “Umbrella” (Uniform) Laplacian

$$\Delta_M^{\text{uniform}} f(\mathbf{v}) = \left(\sum_{\mathbf{v}_j \in N(\mathbf{v})} f(\mathbf{v}_j) \right) - |N(\mathbf{v})| f(\mathbf{v})$$

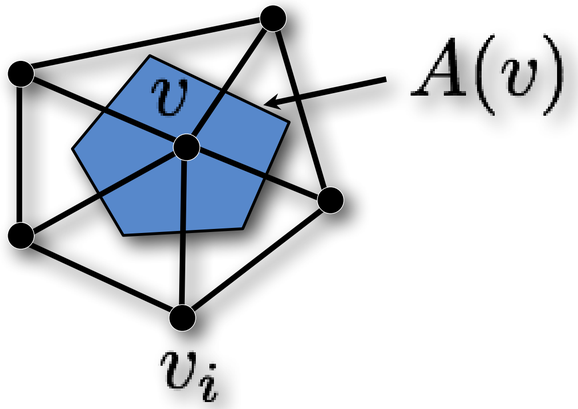
In Matrix Form: $\mathbf{Y} = \mathbf{L}\mathbf{F}$

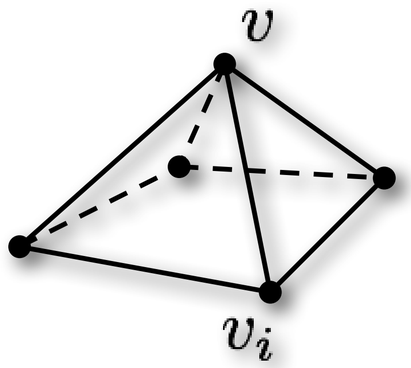
$$\mathbf{Y} = \begin{bmatrix} \Delta f(\mathbf{v}_1) \\ \Delta f(\mathbf{v}_2) \\ \Delta f(\mathbf{v}_3) \\ \cdots \\ \Delta f(\mathbf{v}_N) \end{bmatrix} \qquad \mathbf{L} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1N} \\ w_{21} & w_{22} & \cdots & w_{2N} \\ \vdots & \vdots & \cdots & \vdots \\ w_{N1} & w_{N2} & \cdots & w_{NN} \end{bmatrix} = \{w_{ij}\} \qquad \mathbf{F} = \begin{bmatrix} f(\mathbf{v}_1) \\ f(\mathbf{v}_2) \\ f(\mathbf{v}_3) \\ \cdots \\ f(\mathbf{v}_N) \end{bmatrix}$$

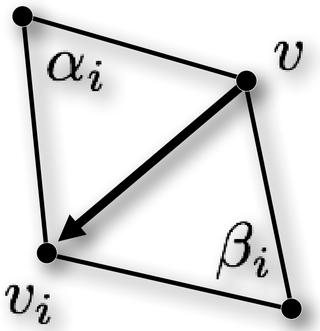
$$w_{ij} = \begin{cases} 0 & i \neq j,\ \nexists \text{ edge } (i,j) \\ 1 & i \neq j,\ \exists \text{ edge } (i,j) \\ -|N(v_i)| & i = j \end{cases}$$

The Cotangent Laplacian

$$\Delta_M^{\text{cotan}} f(\mathbf{v}) = \frac{1}{2A(\mathbf{v})} \sum_{\mathbf{v}_i \in N(\mathbf{v})} (\cot(\alpha_i) + \cot(\beta_i)) \Big(f(\mathbf{v}_i) - f(\mathbf{v}) \Big)$$







- Accounts for edges' differing lengths.
- Converges to the continuous Laplace-Beltrami operator Δ_S with mesh refinement
- Derivation in next recitation lecture.

The Cotangent Laplacian

$$\Delta_M^{\text{cotan}} f(\mathbf{v}) = \frac{1}{2A(\mathbf{v})} \sum_{\mathbf{v}_i \in N(\mathbf{v})} (\cot(\alpha_i) + \cot(\beta_i)) \Big(f(\mathbf{v}_i) - f(\mathbf{v}) \Big)$$

In Matrix Form: $\mathbf{Y} = \mathbf{L}\mathbf{F}$

$$\mathbf{Y} = \begin{bmatrix} \Delta f(\mathbf{v}_1) \\ \Delta f(\mathbf{v}_2) \\ \Delta f(\mathbf{v}_3) \\ \cdots \\ \Delta f(\mathbf{v}_N) \end{bmatrix} \qquad \mathbf{L} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1N} \\ w_{21} & w_{22} & \cdots & w_{2N} \\ \vdots & \vdots & \cdots & \vdots \\ w_{N1} & w_{N2} & \cdots & w_{NN} \end{bmatrix} = \{w_{ij}\} \qquad \mathbf{F} = \begin{bmatrix} f(\mathbf{v}_1) \\ f(\mathbf{v}_2) \\ f(\mathbf{v}_3) \\ \cdots \\ f(\mathbf{v}_N) \end{bmatrix}$$

$$w_{ij} = \begin{cases} 0 & i \neq j,\ \nexists \text{ edge } (i,j) \\ \frac{\cot\alpha_j + \cot\beta_j}{2A(v_i)} & i \neq j,\ \exists \text{ edge } (i,j) \\ -\sum_{v_k \in N(v_i)} w_{ik} & i = j \end{cases}$$

Eigen Sparse Matrix

- Full-sized Laplacian can be huge for large meshes
  - but most elements are zero!
- Instead, only store non-zero elements: **Sparse Matrix**

```
#include <Eigen/Sparse>

Eigen::SparseMatrix<double> Laplacian;
```

Eigen Sparse Matrix

- How to construct a sparse matrix in Eigen:

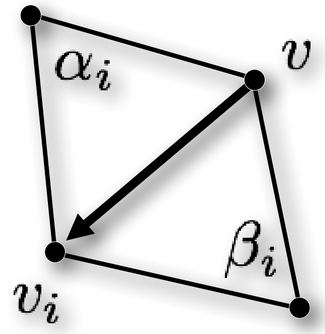
```
//declare size of matrix
L = SparseMatrix<double> (V.rows(), V.rows());

  //declare list of non-zero elements (row, column, value)
std::vector<Eigen::Triplet<double> > tripletList;

//insert element to the list
//if multiple triplets exist with the same row and column, values will be *added*
tripletList.push_back(Eigen::Triplet<double>(source,dest,C(i,e)));

//construct matrix from the list
L.setFromTriplets(tripletList.begin(), tripletList.end());
```

see igl::cotmatrix (V,F)

How to build cotangent matrix

$$\Delta_M^{\text{cotan}} f(\mathbf{v}) = \frac{1}{2A(\mathbf{v})} \sum_{\mathbf{v}_i \in N(\mathbf{v})} (\cot(\alpha_i) + \cot(\beta_i)) \Big(f(\mathbf{v}_i) - f(\mathbf{v}) \Big)$$

- Iterating over vertices is slow
- Instead, iterate over faces and add cot terms to the vertices incident each face edge




```
for i = 1:number_of_faces
 for j = 1:face_valence
  source_vertex = faces(i,j);
  destination_vertex = faces(i,(j+1) % face_valence);
  weight = ....; //laplacian weight for edge (source_vertex, destination_vertex)(cotan or uniform)
  Laplacian(source_vertex, destination_vertex) += weight;
  Laplacian(destination_vertex, source_vertex) += weight;
  Laplacian(destination_vertex, destination_vertex) -= weight;
  Laplacian(source_vertex, source_vertex) -= weight;
 end
end
```

Averaged or Summed?

- We'll see next time that the cotan laplacian we introduced:

$$\Delta_M^{\text{cotan}} f(\mathbf{v}) = \frac{1}{2A(\mathbf{v})} \sum_{\mathbf{v}_i \in N(\mathbf{v})} (\cot(\alpha_i) + \cot(\beta_i)) \Big(f(\mathbf{v}_i) - f(\mathbf{v}) \Big)$$

  computes the **average** Laplacian over some area "A" around v (hence the division by A)
- This gives an **asymmetric matrix**. But if we instead compute the "summed" laplacian (i.e., don't divide by A), we end up with a symmetric matrix.
- This "summed," symmetric Laplacian is more standard in the finite element community, and is related to our Laplacian by a "lumped (diagonal) mass matrix" holding the areas:

lumped mass matrix
(averaging region areas)

$$\mathbf{M} = \begin{bmatrix} A(0) & 0 & \cdots & 0 \\ 0 & A(1) & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & A(N) \end{bmatrix}$$

(L^{averaged} corresponds to formula above)

$$L^{\text{averaged}} = M^{-1} L^{\text{summed}}$$

$$[L^{\text{summed}}]_{ij} = \begin{cases} \frac{1}{2}\Big(\cot(\alpha_{ij}) + \cot(\beta_{ij}) \Big) & \text{if } j \in N(i) \\ -\sum_{j \in N(i)} [L^{\text{summed}}]_{ij} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

Averaged or Summed?

- Suppose we have a linear system:

$$L^{\text{averaged}}\mathbf{x} = \mathbf{b}$$

  i.e. using the asymmetric cotangent Laplacian we introduced earlier.

- Right hand side vector “b”represents a scalar field, storing the desired Laplacian value (**not the summed value!**) at each vertex.

- Solving symmetric sparse linear systems matrices is much more efficient, so we prefer to solve with the summed quantities:

$$ML^{\text{averaged}}\mathbf{x} = \boxed{L^{\text{summed}}\mathbf{x} = M\mathbf{b}}$$

How to compute A(v)

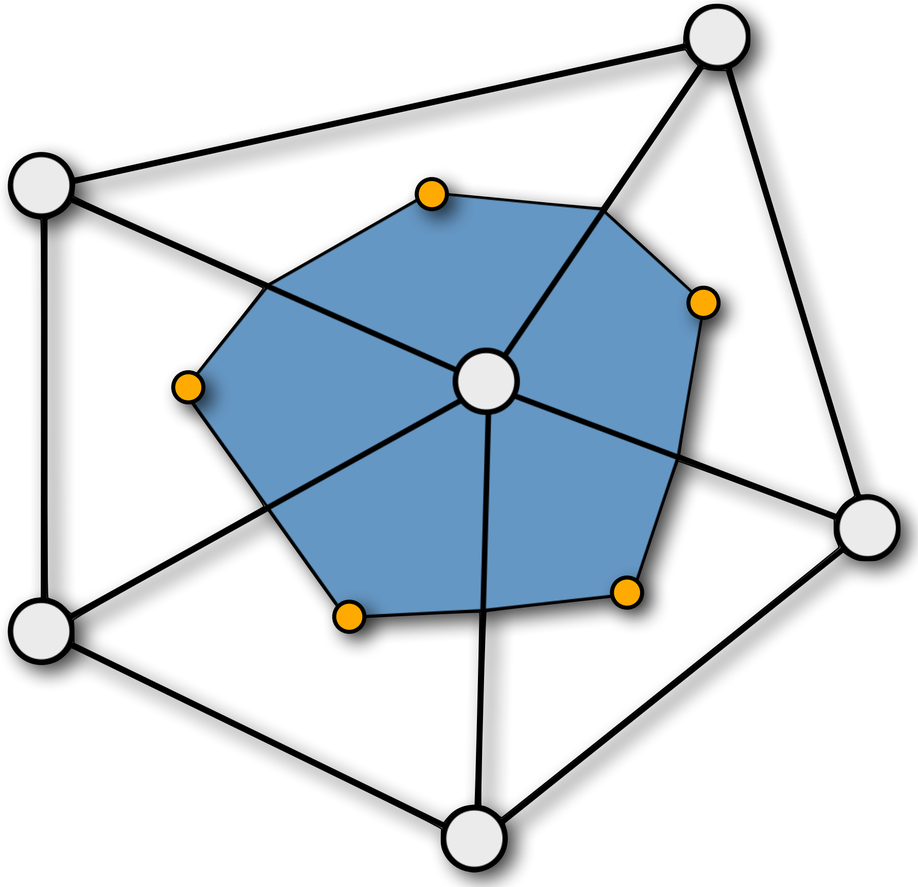
- **Barycentric area**
  - Connect edge midpoints and triangle barycenters
  - Each of the incident triangles contributes 1/3 of its area to all its vertices, regardless of the placement

    + Simple to compute

    + Always positive weights

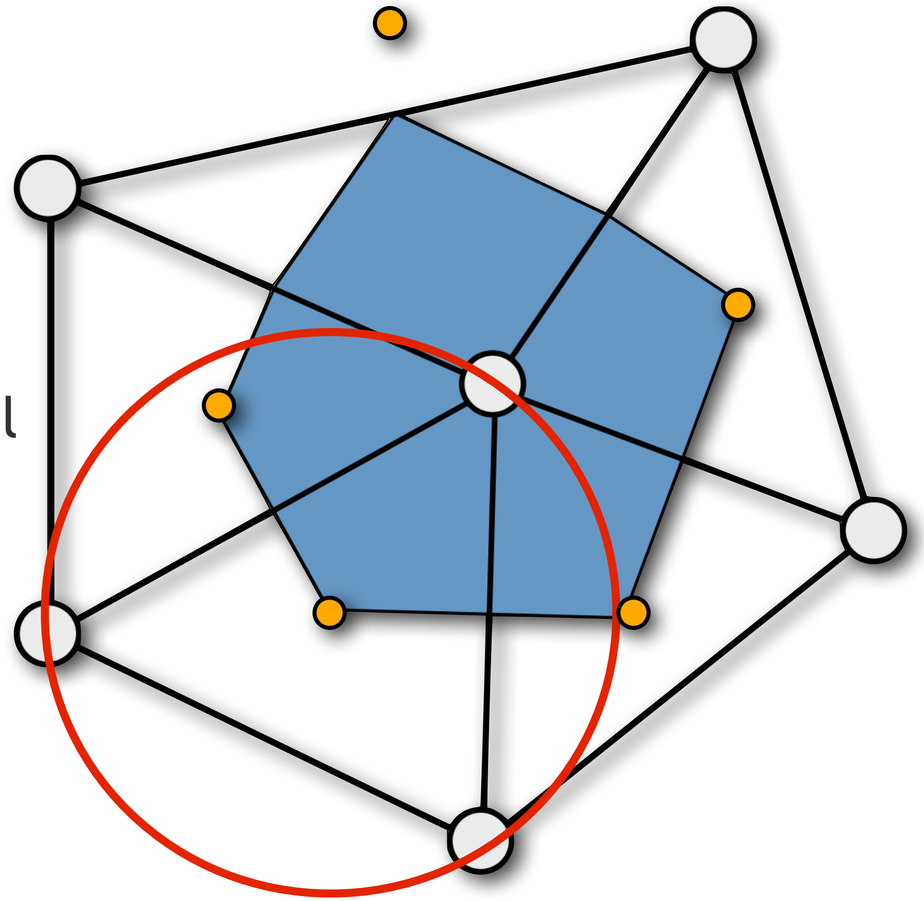
    - Heavily connectivity-dependent

    - Changes if edges are flipped



How to compute A(v)

- **Voronoi area**
  - Connect edge midpoints and triangle circumcenters
  - The 3 resulting quadrilaterals are not equareal
  - Sum contributions from incident triangles



+ Only depends on vertex positioning

- More complicated computations

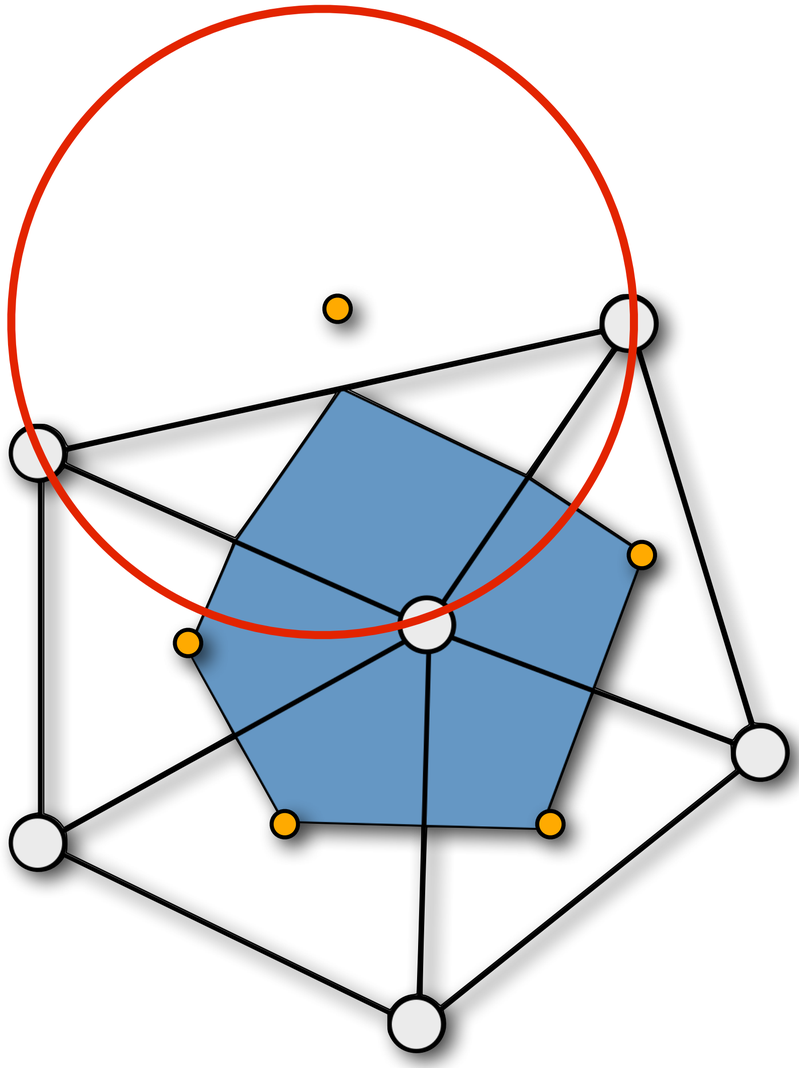
How to compute A(v)

- **Voronoi area**
  - Connect edge midpoints and triangle circumcenters
  - The 3 resulting quadrilaterals are not equiareal
  - Sum contributions from incident triangles
  - For obtuse triangles, circumcenter is outside the triangle -> negative areas!

+ Only depends on vertex positioning

- More complicated computation

- May introduce negative weights (obtuse triangles)

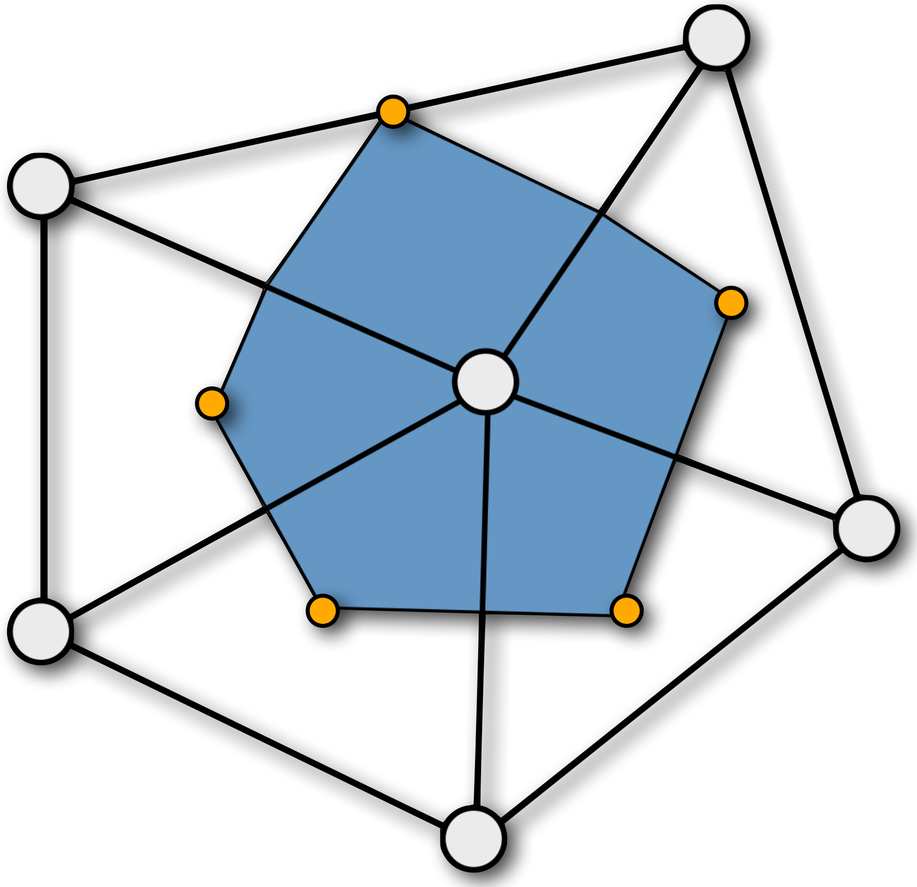


How to compute A(v)

- **Voronoi area - compromise**
  - Connect edge midpoints with:
    - triangle circumcenters , for non-obtuse triangles
    - midpoint of opposite edge, for obtuse ones
  - Sum contributions from incident triangles

+ Only depends on vertex positioning

- More complicated computations



Linear Systems with Eigen

```
#include <Eigen/SparseCholesky>
// ...
SparseMatrix<double> A;
// fill A
VectorXd b, x;
// fill b
// solve Ax = b
SimplicialLDLt<SparseMatrix<double> > solver;
solver.compute(A);
if(solver.info()!=Success) return; // decomposition failed
x = solver.solve(b);
if(solver.info()!=Success) return;  // solving failed
// solve for another right hand side:
x1 = solver.solve(b1);
```

- Cholesky factorization works only for symmetric, positive definite A!
- Other solvers available, e.g. SparseLU:
    - http://eigen.tuxfamily.org/dox/TutorialSparse.html

Questions?